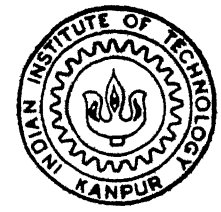
PARALLEL ALGORITHMS FOR TREE PATTERN MATCHING AND GRAPH AUGMENTATION

by

SANJAY NATH



DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY KANPUR

JANUARY, 1994

PARALLEL ALGORITHMS FOR TREE PATTERN MATCHING AND GRAPH AUGMENTATION

A thesis submitted
in partial fulfillment
of the requirements
for the degree of

Master of Technology

by

Sanjay Nath

to the

Department of Computer Science and Engineering

Indian Institute of Technology, Kanpur

January, 1994

CSE-1994-M-NAT-PAR

CERTIFICATE

This is to certify that the work contained in the thesis titled, **Parallel algorithms for tree pattern matching and graph augmentation,** was carried out under my supervision by **Sanjay Nath** and it has not been submitted elsewhere for a degree.

R. K. Ghosh
Associate Professor
Dept. of Comp. Sc. and Engg.
I.I.T., Kanpur
January, 1994

Acknowledgements

I would like to express my deepest gratitude to Dr. R. K. Ghosh for guiding me with infinite patience during the course of this project. At times when the spirit was flagging and confidence shaken, he was there to boost my morale.

Working with him has not only exposed me to the fascinating area of parallel algorithms, but has aroused my curiosity to take a closer look at the complexities involved in the design of parallel algorithms. Once again, I can only say "Thank you, Sir".

Thanks are also due to all my friends here at IIT-K. The more notable among them being Bairagi, Sarkar, Fareed, Sajal, Arush, Nagrajan (our coach), Vivek, Patra and of course, Shreesh.

Contents

List of Figures

Abstract

Three different problems namely, tree pattern matching, biconected augmentation of a graph and triconnected augmentation of outer planar graphs have been studied in this thesis.

Two parallel algorithms for a restricted version of tree pattern matching are proposed. The first one runs in $O(\log n)$ time and uses $O(m)$ processors on a CREW PRAM. The other algorithm has a processor complexity of $O(m/\log m)$ and runs in $O(\log m)$ time on a CRCW PRAM, m and n being the number of nodes in target and pattern trees respectively. The generalized version of the tree pattern matching problem is also considered. The main result for the second problem is an algorithm which runs in $O(\log m)$ time with $O(mn)$ processors on a CREW PRAM, or alternatively in $O(n \log m)$ time with $O(m)$ processors if $\lfloor (m/n) \rfloor \geq 1$.

A parallel algorithm for finding a smallest augmentation to biconnect an undirected graph has been proposed. This algorithm is based on the idea of open ear decomposition. It achieves a time bound of $O(\log n)$ using $O(n)$ processors on a CREW PRAM, where n is the number of vertices in the given graph.

Finally, the work reports a simple approach to triconnect an outer planar graphs using a minimum number of edges. It takes $O(\log n)$ time and $O(n/\log n)$ processors, where n is the number of vertices in the graph.

Chapter 1

Introduction

Trees and graphs provide convenient ways for abstraction, representation of data and modelling of relations and constraints. The problems common among those studied in graph theory are connectivity of graph, testing graph isomorphism under suitable restrictions, shortest paths, graph embeddings, planarity testing, graph traversals and graph augmentations. The modelling of real-life problems may often generate very large graphs. Processing such large graphs may not be feasible even by using the fastest sequential computer under the permissible time constraints. Although the performance of sequential computers has improved steadily over the years, primarily due to improvements in digital hardware technology, the limiting factor is the speed of light in vacuum. While reduction in size and increase of a few orders of magnitude in speed beyond the present levels seem feasible, further improvements in the performance of sequential computers may not be achievable at an acceptable cost. The only way around this bottleneck seems to lie in the use of parallelism in computers. In most cases, a computational task T, such as processing of a graph, can be tackled by partitioning T into a set of smaller subtasks that can be solved simultaneously on a parallel computer. Suppose that a parallel processor P_n can be constructed by combining n copies of a sequential processor P_1. If task T can be partitioned into n subtasks $T_1, T_2, ..., T_n$ of approximately equal size, and P_n can be programmed so that all its n constituent processors execute their subtasks in parallel, then we would expect P_n to execute T up to n times faster than P_1. This potentially higher performance is the main motivation for the introduction of parallelism into computers. However, a price is paid for this increase

in the speed in terms of significant amount of extra hardware. Moreover, since different processors have to communicate with each other and the inter-processor communication being quite costly and dependant on processor inter-connection, parallel algorithms tend to be architecture dependant.

As explained above, by using parallel algorithms, the processing time can be reduced by a substantial factor. Therefore, it should be possible to solve larger instances of a problem under the permissible constraints of time through a parallel algorithm as opposed to a sequential one.

We investigated the problem of tree pattern matching *TPM* for parallel computation[1]. Tree pattern matching is a restricted version of the subtree isomorphism problem, which is known to be in random *NC* [14]. Two different versions of pattern matching for trees have been considered. For the restricted version of *TPM*, we have suggested three different approaches, tree merging, string matching and tree splitting. Tree merging technique leads to an $O(\log n)$ time, $O(m)$ processor algorithm. The string matching approach yields a cost-optimal algorithm running in $O(\log m)$ time with $O(m/\log m)$ processors. The idea of tree splitting is a novel approach which leads to a cost-optimal algorithm. The problem which was investigated next, relates to design of efficient parallel algorithms for finding smallest graph augmentations. Augmentation of a graph means insertion of a minimum number of edges to a graph, so that the resulting augmented graph satisfies some connectivity property. The particular problems of augmenting a graph to make it biconnected or triconnected have been studied in this thesis. Our main result is an algorithm for biconnecting a graph. It has a time complexity of $O(\log n)$ using $O(n)$ processors[2]. A parallel solution for finding smallest augmentation to triconnect a graph was considered. The effort did not bear much fruit due to paucity of time, but the basic framework for an approach has been identified. Subsequently, a simpler problem of triconnecting an outer planar graph using a minimum number of edges was solved. A cost optimal parallel algorithm for this problem has been proposed.

[1]Nath S. and Ghosh R.K., Parallel algorithms for tree pattern matching, Appeared in *Proc. 3rd National Seminar on Theoretical Computer Science*, May 1993, pp. 299-308.

[2]Nath S. and Ghosh R.K., Parallel algorithm for biconnected augmentation, Appeared in *Proc. 17th National Systems Conference*, December 1993, pp. 529-532.

1.1 Models of Computation

The organzation of a computer can be classified depending upon the number of instruction and data streams used by the system into the following four classes:

1. Single Instruction Single Data stream (SISD).
2. Single Instruction Multiple Data stream (SIMD).
3. Multiple Instruction Single Data stream (MISD).
4. Multiple Instruction Multiple Data stream (MIMD).

SIMD and MIMD are the most widely used computer organizations which support parallel computation. An SIMD computer consists of many identical processors. Each of these processors possess its own local memory where it can store data. All the processors operate under the control of a single instruction stream issued by a central control unit. The processors then handle different data streams. The inter-processor can be either via a shared memory or an inter-connection network.

An SIMD computer, where data sharing is through a common bank of memory accessible to all the processors is the most widely used theoretical model for parallel computation. Depending on the extent of data sharing, such models can be classified further into four subclasses:

1. Exclusive Read Exclusive Write (EREW).
2. Concurrent Read Exclusive Write (CREW).
3. Exclusive Read Concurrent Write (ERCW).
4. Concurrent Read Concurrent Write (CRCW).

The models are listed in an ascending order on power (speed) of computations.

1.2 Algorithmic Performance

The performance of a parallel algorithm is judged by its running time, the number of processors used and the cost. The running time of an algorithm is the time taken to compute the results. A theoretical analysis of the running time is done by counting the number of basic operations executed by the algorithm in the worst-case.

Another criterion in evaluating a parallel algorithm is the number of processors required. The cost of a parallel algorithm is defind as the product of the parallel running time and the number of processors used. A cost optimal parallel algorithm is one for which the cost equals the worst-case time of the fastest sequential algorithm. A related term efficiency is defined as the ratio of the worst-case time of the fastest sequential algorithm to the cost of parallel algorithm.

$$Efficiency = \frac{worst\text{-}case\ time\ of\ the\ fastest\ sequential\ algorithm}{cost\ of\ parallel\ algorithm}.$$

1.3 Thesis outline

This thesis is organized into 5 chapters including the introduction. In chapter 2, two parallel algorithms for a restricted version of tree pattern matching are proposed. The first one runs in $O(\log n)$ time and uses $O(m)$ processors on a CREW PRAM. The second algorithm has a processor complexity of $O(m/\log m)$ runs in $O(\log m)$ time on a CRCW PRAM, m and n being the number of nodes in target and pattern trees respectively. The generalized version of the tree pattern matching is also considered. The main result for the second problem is an algorithm which runs in $O(\log m)$ time with $O(mn)$ processors on a CREW PRAM.

In chapter 3, the problem of augmenting a graph using a minimum number of edges to make it biconnected is considered. The algorithm developed for the purpose has a time complexity of $O(\log n)$ using $O(n)$ processors, n being the number of nodes in the given graph.

The problem of triconnecting an outer-planar graph by adding a minimum number of edges is considered in chapter 4.

Finally, chapter 5 concludes the thesis, giving a brief summary and a discussion of some of the interesting open problems encountered during our work.

Chapter 2

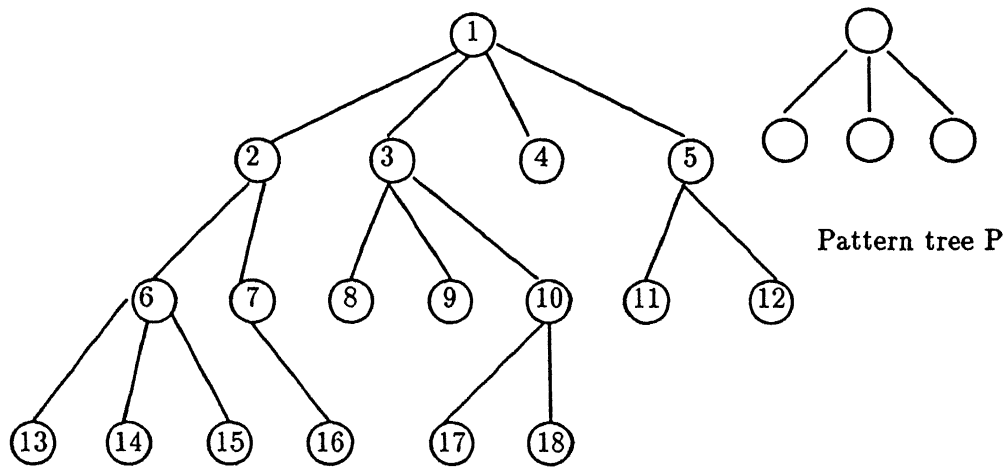
Tree Pattern Matching

2.1 Introduction

Tree patterns find wide spectrum of applications in the area of computer science. Many computing techniques involve simplifying expressions (trees) by repeatedly replacing special types of subexpressions (subtrees) according to a set of replacement rules. Tree replacements may be used in automatically generated interpreters for non-procedural programming languages. Intermediate code produced by compilers may be represented by trees. Certain types of code-optimizations, such as elimination of redundant operations and constant propagation may be viewed as replacement rules in trees. In automatic proving of equational theorems - a set of equational axioms may be treated as replacement rules and one side of the equation could be transformed to the other side using a sequence of tree replacements.

The *Restricted Tree Pattern Matching problem (RTPM)* takes two trees, a pattern tree P with n nodes and a target tree T with m nodes as input, where $m \geq n$. It finds all nodes $n_i \in T$ such that the entire subtree rooted at n_i matches with the tree P. The *Tree Pattern Matching Problem (TPM)* considers ordered labelled trees in which the edges are labelled according to the rank of the node among its siblings. The pattern tree P matches T at node $n_i \in T$ such that :

1. The root of P maps to n_i



Target Tree T

Solution for RTPM {6}

Solution for TPM {1, 3, 6}

Figure 2.1: Illustration of *RTPM* and *TPM*

2. If $x \in P$ maps to $y \in T$ and x is not the root of P, then x and y have identical ranks among their siblings

3. If $x \in P$ maps to $y \in T$ and x is not a leaf, then the i-th child of x maps to the i-th child of y

Given trees P and T of sizes n and m respectively, we wish to compute the set of nodes of T at which P matches. In both *RTPM* and *TPM*, the nodes $n_i \in T$ are called match_nodes with respect to the pattern tree P. Two parallel algorithms for the subtree isomorphism are proposed. The first algorithm runs in $O(\log n)$ time and uses $O(m)$ processors on a CREW PRAM, whereas the second algorithm has a processor complexity of $O(m/\log m)$ and runs in $O(\log m)$ time on a CRCW PRAM.

The main result for the *Tree Pattern Matching (TPM)* is an algorithm which runs in $O(\log m)$ time with $O(mn)$ processors on a CREW PRAM, or alternatively in $O(n \log m)$ time with $O(m)$ processors if $\lfloor (m/n) \rfloor = l$, where l is an integer ≥ 1. Illustrations of both RTPM and TPM are given in Figure 2.1.

Parallel algorithms for RTPM have been independently studied by Grossi [15] in a different context. Grossi applied string matching over representation with strings containing

don't care symbols. We studied three different approaches for design of simple as well as efficient parallel algorithms for RTPM. The first algorithm for RTPM is based on tree merging technique. It runs in $O(\log n)$ time with $O(m)$ processors on CREW PRAM. The second algorithm uses parallel matching algorithm [36] and is somewhat similar to Grossi's [15] approach. The difference is in evolving the scheme for coding trees. We use Euler tour to find a unique well balanced sequence of parentheses to code a tree and its subtrees. This coding can enable to check isomorphism with data-items associated with each node. The third parallel algorithm is the simplest among the three algorithms. We outline steps of this algorithm in chapter 5.

The problem of term matching [33], which is a special case of unification, together with forest matching is closely related to the TPM. The first optimal speedup algorithm reported in [21] can be applied in parallel for each subtree of T to solve TPM. The resulting time and processor complexities will be $O(\log m)$ and $O((m^2+n)/\log m)$ respectively. However, the algorithm in [21] for CREW PRAM involves complex processor scheduling, randomization scheme [31] and universal hash functions [2] for simulating concurrent writes, as their parent algorithm is designed for a CRCW model of computation. In our algorithm the basic ideas are quite simple. Furthermore, it runs in $O(\log m)$ time with $O(mn)$ processors on a CREW PRAM.

2.2 Restricted Tree Pattern Matching

In this section we describe two algorithms for the restricted version of tree pattern matching. Tree merging forms the basis for the first algorithm, whereas the second algorithm uses Vishkin's [36] string pattern matching technique and is cost optimal.

2.2.1 Tree Merging Approach

In the tree merging approach for Restricted Tree Pattern Matching , the target tree T is grown in k stages where $k = \lceil \log h(P) \rceil + 1$ where h(P) denotes height of P. In the worst case h(P) = $\Omega(n)$. For each node $v \in T$ the subtree rooted at v is grown stage by stage. Tree merging is illustrated in Figures 2.3, 2.4, 2.5 for a sample target tree T, shown in figure 2.2.

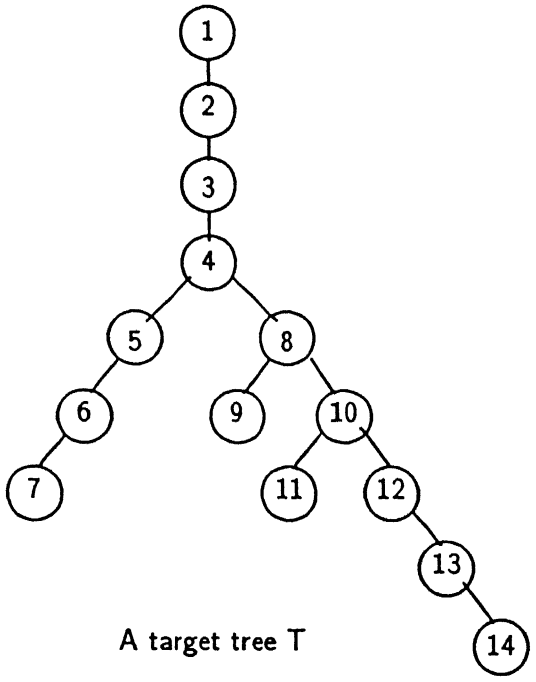


Figure 2.2: A sample target tree T

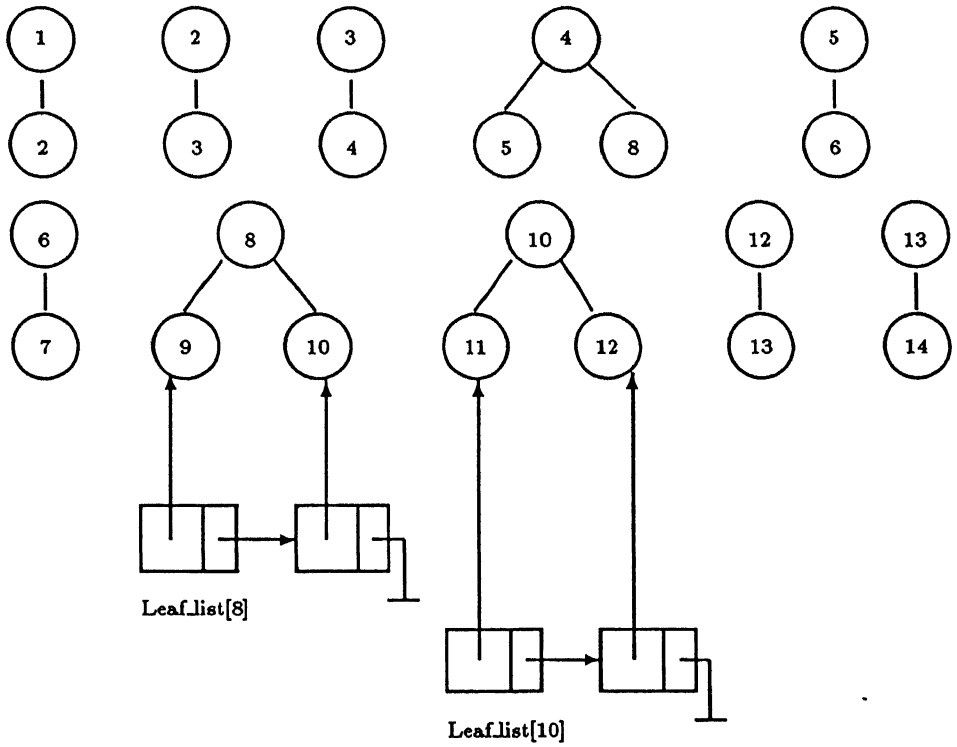


Figure 2.3: Stage 1 of Tree-merging

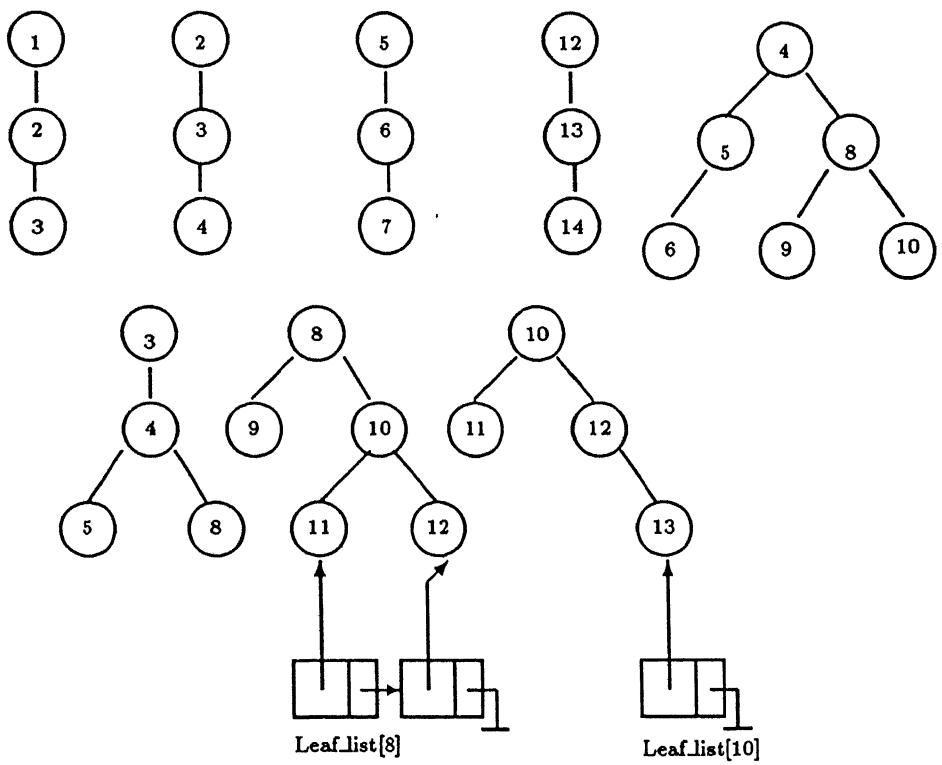


Figure 2.4: Stage 2 of Tree-merging

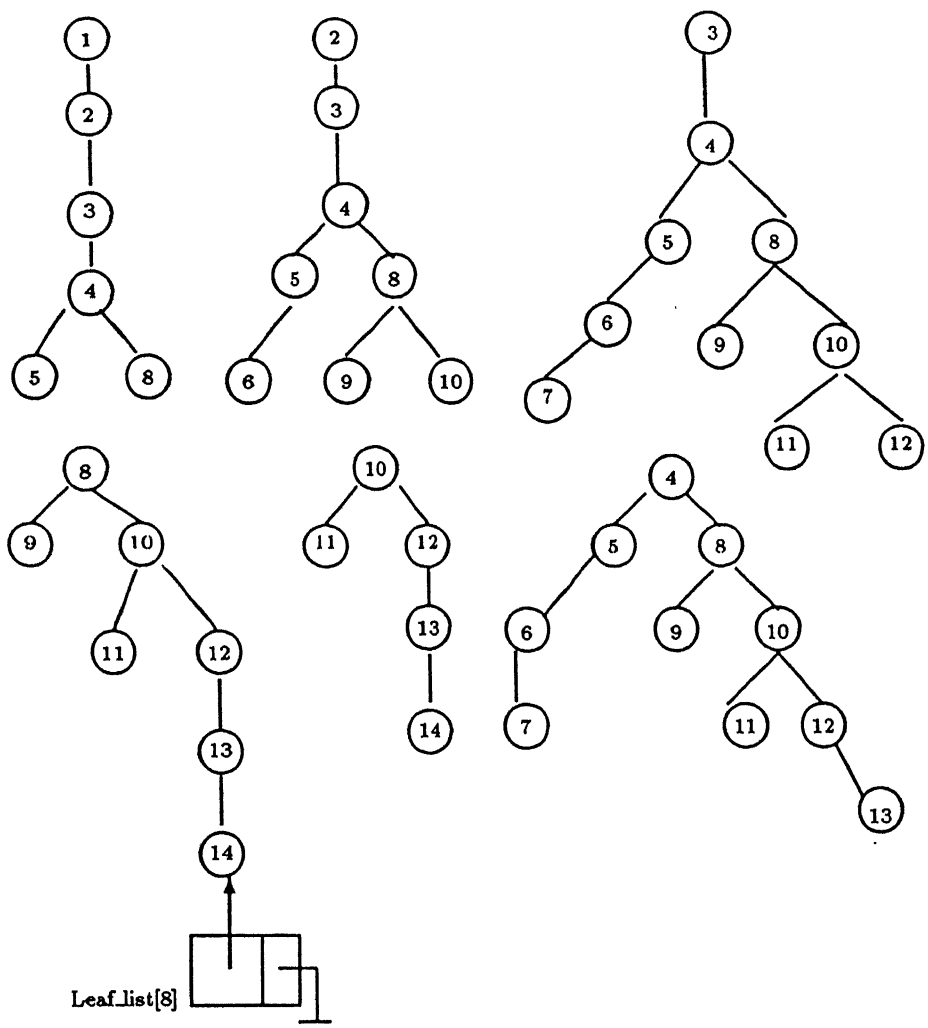


Figure 2.5: Stage 3 of Tree-merging

After stage i, the height of the subtrees can be at most 2^{i-1}. The actual matching of P with the subtrees can be converted to a problem of position wise matching of a sequence of well balanced parentheses with another sequence of well balanced parenthesis starting with the first position of the target sequences. The pattern tree P can be converted into such a balanced sequence by using the well known Euler-Tour technique on trees. The same is to be done for all the subtrees of T generated after $\lceil \log n \rceil$ stages. The details regarding this conversion are given in Section 2.2.3. In general, all the subtrees for the first stage can be generated in $O(\log n)$ time using $O(m/\log n)$ processors. The crucial part is to arrive at the subtrees for $i+1$-th stage from the subtrees generated in the previous stages in $O(1)$ time. This is based on the claim stated in Lemma 2.1.

Lemma 2.1 *A node $v \in$ T, where v is not a leaf node, can be a leaf node for at most one of the subtrees so far generated.*

Proof: Let v be a terminal node in two distinct subtrees T_1 and T_2 formed at stage i. Let r_1 and r_2 be the roots of T_1 and T_2 respectively. Then, $2^{i-2} < |r_1 \xrightarrow{*} v| \leq 2^{i-1}$ and $2^{i-2} < |r_2 \xrightarrow{*} v| \leq 2^{i-1}$. Also, both r_1 and r_2 are ancestors of v in T. If $|r_1 \xrightarrow{*} v| = |r_2 \xrightarrow{*} v|$, then $r_1 = r_2$, because v cannot have two distinct ancestors at the same height in T. Therefore, we assume without loss of generality that $|r_1 \xrightarrow{*} v| > |r_2 \xrightarrow{*} v|$. This implies that $|r_2 \xrightarrow{*} v| < 2^{i-1}$. It is also known that v is an internal node of T. Note that when tree merging is carried out, a node at height $< 2^{i-1}$, where i is the stage number, can only be a leaf node of T. The above observation implies v is not a terminal in T_2. ■

From Lemma 2.1, it follows that the subtrees for the i+1-th stage can be generated by simply hanging the latest subtree generated, rooted at say y in T, at the position in the subtree for which y is a leaf node.

The input to the algorithm is in the form of a structure for each vertex of the target tree T defined formally below in C language :

```
struct child {
    int child_no.; /* # of children */
    int rank ; /* rank among its siblings */
    int * list_ptr }; /* pointer to the associated array list */
```

Node_array	child_no.	rank	list_ptr	lists
1	1	Null		2 0
2	1	1		3 1
3	1	1		4 2
4	2	1		5 8 3
5	1	1		6 4
6	1	1		7 5
7	0	1		6
8	2	2		9 10 4
9	0	1		8
10	2	2		11 12 8
11	0	1		10
12	1	2		13 10
13	1	1		14 12
14	0	1		13

Complete input tree representation

Figure 2.6: Data structure for tree T

node_array * child[MAX]; /* MAX = m */

An array, list, is associated with each of the nodes i and the size of the list is equal to number of children of i plus 1. The k-th child of i is stored at position list[k], where the list we are referring to here is the one associated with node i. The parent of a node i is located at position node_array[i] $\rightarrow$ list[node_array[i] $\rightarrow$ rank + 1]. We assume that this data structure is available to us as the input to all the algorithms presented here. (see Figures 2.2, 2.6).

2.2.2 Algorithm for tree merging

The steps of this approach for tree merging are outlined below.

Step 1: [Marking]

Mark all the non-leaf nodes of T for growing. These marked nodes only are considered for further growth.

Step 2: [Building Stage 1]

forall $i : 1 \leq i \leq m$ do

if (i is not a leaf node) then {

declare children of i as leaves for subtree with root i;

put a pointer to node i in list_array[i] ; }

else {

put a pointer to parent of i ; }

Do step 3 for $\lceil \log h(P) \rceil + 1$ times.

Step 3: [Merging]

Add the leaf nodes of the subtrees being hanged to the subtrees to which they are hanged.

Comment: Note that step 3 only keeps track of the leaf nodes of a subtree. By leaf nodes of a subtree, we mean the nodes of that subtree which are at a depth of 2^{i-1}, if i is the current stage number. The actual merging of subtrees is quite simple. Just the parent pointer of a root of the subtree being hanged is to be suitably modified.

Step 4: [Postprocessing]

Check whether a subtree can grow further, i.e., whether all its terminal nodes are indeed leaf nodes of T. If not, rule out the subtrees rooted at that node for matching against P.

Comment: This ensures that a node of T is present in at most one subtree out of those which pass this postprocessing step. Thus, the total number of nodes in all the subtrees to be considered for actual matching with P is $\Omega(m)$.

Step 5: [Coding of subtrees]

Assigning $k_j / \log n$ processors to every subtree, call algorithm Code as given in the next section simultaneously, where k_j is equal to the number of nodes in a subtree.

Comment: Note that the number of nodes, k_j, in a subtree can be found out easily.

Step 6: [Matching]

Match the euler-sequences of the subtrees against that of P simultaneously.

Since Step 3 is where the actual tree merging takes place, it may be interesting to work out the details of this step. A subtree t_j with l_j leaf nodes, is assigned l_j processors. Each processor is supplied with

1. A pointer to the head node and a pointer to the tail node of the list of leaf nodes of the subtree rooted at that node of t_j, which are given the names list_head and list_tail respectively.

2. A pointer to the leaf node assigned to it and a pointer to the next node in the list of leaf nodes. These will be named as header and tail respectively.

```
Do in parallel {
    R_head = list_head;
    /* R_head and R_tail are temporary variables */
    R_tail = list_tail;
    header.data = R_head.data;
    header.next = R_head.next;
    list_array[R_head_data] = header;
    dispose(R_head);
    R_tail.next = tail;}
```

As is clear, this operation will take $O(1)$ time with a total of $O(m)$ processors.

Theorem 2.1 *The algorithm based on tree merging has a time complexity of* $O(\log n)$ *and takes* $O(m)$ *processors.*

Proof: As is clear, step 3 itself takes $O(m)$ processors and $O(\log h(P))$ time, where h(P) = height of P. The matching part can be done in $O(\log n)$ time with $O(m)$ processors. Thus, the overall time complexity is $O(\log n)$ with $O(m)$ processors. ■

2.2.3 String Matching Approach

Fast parallel algorithms for string matching problems can be found in literature. An $O(\log^* n)$ time parallel algorithm using optimal number of processors on a CRCW PRAM has been proposed by Vishkin [35] where n is the size of the text string.

If a tree can be coded uniquely (and appropriately) by a string of symbols, RTPM reduces to a simple substring match problem. In other words, RTPM can be decomposed into two distinct phases, viz.,

- obtain an "appropriate" string representation of the trees P and T.
- apply string matching to locate all occurrences of n_i in T where pattern P matches.

Since the above problem deals with subtrees, it is worth noting that the property of the appropriate representation, referred to above, must have a depth-first property. It means that the scheme for representation must ensure that the string corresponding to any subtree of a node must occur as a proper substring of the string representing the whole tree. This is an essential condition so that the pattern string occurs as a contiguous substring in the target string. Before presenting formal details of our coding scheme for representing trees in form of a string, the theoretical background which forms the basis of this scheme is introduced below.

An *Ordered Tree* is a directed tree such that for each internal node, there is a defined order of its children. An *Ordered Forest* is a sequence of ordered trees.

Lemma 2.2 *There is a one to one correspondence between all ordered forests with n nodes and the well-formed sequences with n pairs of parenthesis. The number of such ordered forests (and the well-formed sequences) is denoted by $C(n)$, Catalan number, where $C(n) = \frac{1}{n+1}{}^{2n}C_n$.*

Proof: See [10]. ∎

Our basic approach uses Lemma 2.2. An ordered tree with n nodes can always be decomposed into a root node and an ordered forest comprising of subtrees of the root node. The ordered forest with $(n-1)$ nodes yields a well-balanced sequence of $(n-1)$ pairs of parentheses. This entire sequence when enclosed within a pair of parenthesis gives a well-balanced sequence of n pairs. It can easily be seen that such a sequence corresponding to any ordered tree is unique applying Lemma 2.2.

It is worth noting that the correspondence between ordered trees and well-formed sequence is a bijective mapping. This forms the basis of the representation scheme used in our algorithm CODE().

The trees are represented by using symbols '(' and ')', one pair for each node of a tree. The coding for a tree is thus a well-formed sequence of n pairs of parentheses. The coding can be defined recursively as :

$$Code(T[x]) == \begin{cases} \text{'()' if } T[x] \text{ is a single node} \\ Concat(Code(x_1)Code(x_2)....Code(x_k)) \text{ otherwise} \end{cases}$$

here, node x has k children and x_i denotes the i-th child of node x. Euler tour of the tree is used to implement the coding scheme for trees. Note that, Euler tour starts from a node, traverses each edge exactly once in both direction and finally returns to the starting node. An algorithm for implementation of the coding mechanism is outlined in Algorithm CODE().

Algorithm CODE(T,n)

/* Inputs - A Tree T with n nodes*/

/* Output - String of parenthesis */

1. For all edges (i,j) of the tree T do in parallel
   Tournext((i,j)) = next((j,i));
2. For all edges (i,j) of the tree T do in parallel
   if (j == parent(i))
   mark type((i,j)) as Backward
   else
   mark type((i,j)) as Forward
3. Call List-Ranking on the obtained sequence of next
   pointers to obtain the rank of each edge (i,j) in the tour.
4. For each edge (i,j) do in parallel
   if (type((i,j)) is Backward)
   str[rank((i,j))+1] = ")"
   node[rank((i,j))+1] = NULL
   else
   str[rank((i,j))+1] = "("

node[rank((i,j))+1] = j

5. str[1] = "("; node[1] = 1
   str[2n]= ")"; node[2n]= NULL
6. return str

In the Algorithm CODE() Step 1 finds the Euler tour for the tree T. The tour is obtained as a 'linked-list' of the edges - the pointers pointing to the next edge in the Euler tour. Step 2 labels each edge in the tour as forward or backward - since every edge in the tree has two edges in the tour obtained - one in either direction. The list representation of the tour is converted to an array representation by using List Ranking. In steps 4 and 5 the edge-characterization of the tour is converted to a node- characterization which is the way Code() has been defined. The tour of length $2(n-1)$ is converted to a coding of length $2n$ by prepending and appending the characters '(' and ')' respectively. It is worth noting that the edges $(p(v), v)$ and $(v, p(v))$ correspond to node v. Step 3 in the Algorithm CODE() involves a List ranking and can be implemented in $O(\log n)$ time using $O(n/\log n)$ processors. All other steps can be trivially achieved in $O(\log n)$ time using $O(n/\log n)$ processors.

Lemma 2.3 *Algorithm CODE() runs in $O(\log m)$ time using $O(m/\log m)$ processors.*

Algorithm CODE() has to be applied to the pattern tree P and the target tree T. Hence Coding will take $O(\log m)$ time and $O(m/\log m)$ processors. The subtree isomorphism can now be tested by applying a parallel string matching algorithm (Refer [35]). The list of nodes n_i is formed by using array node, associated with the string.

2.3 Parallel Algorithms for TPM

Several sequential algorithms have been proposed for TPM, see [8, 17, 23]. $O(nm)$ time remained a bottleneck for the above problem for a long time. Recently, Kosaraju in [23] gave a serial algorithm running in $O(mn^{0.75}polylog(n))$. This was further improved to a bound of $O(mn^{0.5}polylog(n))$ in [8].

Note that m is the number of nodes in the target tree and hence each node could be a possible candidate for potentially becoming a match-node in the general matching problem.

2.3.1 Algorithm TPM_1

Algorithm TPM_1() describes the first algorithm for the above problem which has a logarithmic time complexity. Figure 2.6 illustrates a 'complete' representation of a tree. A number giving rank of every edge emanating from a node is associated with each edge of the tree.

Definition 2.1 *path(v,u) is a sequence of positive integers, specifying the unique path from v to u (v is an ancestor of u) recursively, as follows:*

1. *if v is the parent of u, then path(v,u) = i iff u is the i-th child of v ($i > 0$).*
2. *if u' is the i-th child of u and path(v,u)=(k1,k2,...kl), where l= depth(u)-depth(v), then path(v,u') = (k1,k2,...,kl,i).*

Note that the string path(v,u) is a prefix of the string path(v,x), where u is an ancestor of x. We shall refer to this property as the prefix property.

Lemma 2.4 *For any induced subtree $T[v_i]$ of the tree* T *there is at most one node v_j such that path(v_i,v_j) is isomorphic to some path(r,v_k) belonging to the pattern tree* P. *Converse of the argument also holds. Here, r denotes the root of the pattern tree* P.

Proof: This is clear from the fact that every path of a tree or a subtree rooted at any node is unique. ■

Lemma 2.5 *A node $v \in T$ is a match-node with respect to a pattern tree P iff node v has exactly n nodes (where n is the size of* P*) in its subtree which have isomorphic paths to the paths corresponding to every node in T.*

Proof: From the definition of TPM it is clear that in order to find a match-node $v \in T$ we need to get images for every node of P with a distinct node. From Lemma 2.4, it follows that the image of each node of P in T is unique. Thus any node $v \in T$ is a match-node iff every node in P gets an image in the subtree of T rooted at v. Therefore the number of isomorphic substrings is exactly equal to the number of nodes in P, i.e., n which is the size of P. Note that the root of P has a path Null. ■

Algorithm TPM_1()

/* Input - Trees P and T of sizes n and m respectively */

/* Output - List of match nodes in the target tree T */

1. for each edge of the tree P and T do in parallel

   Associate a number, rank[(i,j)].

/* rank[(i,j)] gives the rank of the son j of the parent node i */

2. for each node $v_i \in P$ do in parallel

   Store the string (in $Pstr_i$) of numbers corresponding

   to rank[] of each edge on the path($r \rightarrow v_i$) in P.

3. for $1 \leq i \leq m$ do in parallel

   for $1 \leq j \leq m$ do in parallel

   count[i][j] = checkmatch(v_i, v_j)

   /* Function checkmatch() returns 1 or 0 */

4. for $1 \leq i \leq m$ do in parallel

   count[i][$m+1$] = $\sum_{j=1}^{j=m} count[i][j]$

5. for $1 \leq i \leq m$ do in parallel

   if (count[i][$m+1$] == n)

   append node i in the list L_i

6. return L_i

checkmatch(v_i, v_j)

/* Checks if with root as v_i, node v_j has some corresponding node in P*/

1. Store the string (in $Tstr_j$) of numbers corresponding

   to rank[] of each edge on the path(v_i, v_j) in T.

2. for $1 \leq k \leq n$ do in parallel

   if ($Pstr_k == Tstr_j$)

   return 1

3. return 0

Step 1 of the procedure checkmatch() can be implemented by the ancestor table computation and can be easily implemented in $O(\log n)$ time using $O(n^2/\log n)$ processors [34]. In step 2 there are n parallel executions of string-matching and hence can be achieved using $O(\log n)$ time and $O(n^2/\log n)$ processors. Therefore, checkmatch() can be implemented in $O(\log n)$ time using $O(n^2/\log n)$ processors. The bottleneck step in the main Algorithm TPM_1() is the step 3 which involves m^2 parallel calls to checkmatch(), hence requiring $O(m^2n^2/\log n)$ processors and taking $O(\log n)$ time. Steps 1 and 2 which are the pre-processing steps have costs $O(m)$ and $O(n)$ respectively which are within the resource bounds of step 3. Step 4 involves m executions of parallel summing and hence can be implemented in time $O(\log n)$ and $O(n^2/\log n)$ time. Finally, step 5 can trivially be implemented in $O(1)$ time using $O(m)$ processors.

Lemma 2.6 *Algorithm TPM_1() can be implemented in* $O(\log n)$ *time using* $O(m^2n^2/\log n)$ *processors.*

2.3.2 Algorithm TPM_2

In the level order numbering of a complete binary tree, each node numbered as i has a leftchild numbered as $2i$ and a rightchild numbered as $2i + 1$. Equivalently, the root is numbered as 1 and the parent of each node $i > 1$ is numbered by $\lfloor i/2 \rfloor$.

With the above background the basic steps of the algorithm TPM_2 can be outlined as follows.

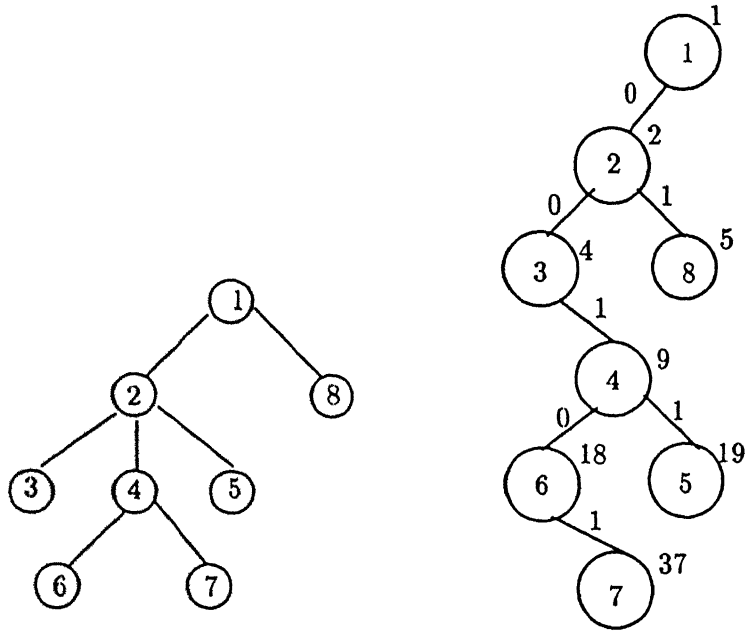
Algorithm TPM_2 ()

Step 1: [Conversion into a binary tree.]

Assign one processor to each node of T. Assuming nodes of T are numbered 1,2,....m with 1 as the root. We shall refer to the processor assigned to node i as $P_i(1 \leq i \leq m)$.

Comment: The output of this step will be a binary tree of the following structure :

```
struct b_node {
    struct b_node * r_child; /* pointer to the right child */
    struct b_node * l_child; /* pointer to the left child */
    struct b_node * parent ; }; /* pointer to the parent */
```



An input tree and its corresponding renumbered binary tree

Numbers outside the nodes refer to the numbers obtained after renumbering

Figure 2.7: Renumbering of nodes of a Tree

```
struct b_node * b_tree[MAX]; /* MAX = m */
```

Step 2: [Renumbering the nodes of the binary tree]

Renumber the nodes of the resulting binary tree in a level order fashion as shown in Figure 2.7.

Comment: Each processor associated with a node other than root of target tree T' does the following steps 2.1 through 2.2 in parallel.

Step 2.1: [Labelling]

Label the incoming edge from its parent with a 0 or 1 depending upon whether the node is a left or a right child respectively.

Step 2.2: [Pointer Doubling]

Apply standard pointer doubling technique [13] to reach the root and in this process accumulates the bits encountered on the edges. Note that depth of a node etc. is also available in the form of the length of the bit vector. A bit 1 is added as the M.S.B. in all the bit vectors.

Step 2.3: [Generating m sequences]

Now assign a processor to every node of P and generate the m sequences each consisting of n binary integers.

Simply prepend the bit representation of node $n_i \in T'$ to the bitstring obtained by XOR-ing bitstring for $2^{h(P')}$ with the bit representation of node $m_j \in P'$. For convenience, convert all the binary integers into decimals. This poses no problem since the maximum size of the bit vectors is m, it can be done in $O(\log m)$ time.

Step 3: [Sorting of renumbered nodes of T']

Sort the decimal numbers corrsponding to renumbered nodes of T' in increasing order.

Comment: Note that step 3 can be done in $O(\log m)$ time with m processors [4].

Step 4: [Sorting m sequences generated for P'].

Sort each of the m sequences in increasing order. This requires mn processors and $O(\log n)$ time.

Comment: Note that the renumbered leaf nodes of P' need to be sorted only once. Thereafter, the positions of leaf nodes in the m sequences get fixed and no repeated sorting is required.

Step 5: [Binary search]

Using binary search each processor tries to find out whether the number assigned to it is present in the target tree sequence.

Comment: If all processors assigned to say k-th sequence report success, a match node has been found.

Lemma 2.7 *Consider the renumbering of nodes at Step 2. If the binary representation of a renumbered node is known, then the renumber of its left child is given by appending a 0 as the L.S.B. and that of its right child by appending an 1 as the L.S.B.*

Proof: Appending a 0 is essentially shifting left by a bit which is equivalent to multiplying by 2. Similarly appending a 1 is equivalent to multiplying by 2 and adding 1 to the original number. Therefore, the renumbering is consistent with level order numbering scheme of nodes of a complete binary tree. ■

Lemma 2.8 *The lengths of all the bit sequences that are needed to handle at any stage in the algorithm TPM_2 are bounded above by m.*

Proof: Only those nodes v of T' are candidates for match nodes for which the relation (height(T') - depth(v)) $\geq$ height(P') holds. When this relation is barely satisfied, depth(v) is maximum. This in turn implies that the bit representation of v is longest. Therefore, bit representation of candidate nodes will never exceed height(T') - height(P') bits. Hence the bit juxtaposition step will never yield binary numbers of length greater than height(T') which in turn is bounded by m. ■

Lemma 2.9 *A leaf node $v \in$ P (T) is a leaf node of P' (T') only if it is the rightmost son of its parent.*

Proof: It is quite clear from the way a general tree is converted to a binary tree that a leaf node which has a right sibling is no longer a leaf node in the corresponding binary tree. Hence, the proof follows. ■

Lemma 2.10 *If a leaf node of P' is found in the sorted sequence of renumbered nodes of T' then all ancestors of v are also present in the sorted sequence of renumbered nodes of T'.*

Proof: The way the nodes of P' and T' have been renumbered, it is quite clear that the binary number corresponding to ancestors of a node v is a prefix of the binary number corresponding to the node v. Hence, searching only for the leaf nodes of P' will suffice. ■

Lemma 2.11 *If $\lfloor m/n \rfloor = l$, then steps 4 and 5 can be done in done in $O(n \log m)$ time with m processors.*

Proof: Clearly, we can assign at least l processors to every leaf node v of P'. Let them be numbered $P(v,1), P(v,2)....P(v,l)$. They make at most $O(n)$ passes taking (ij + 1) -th node to (i + 1)j -th nodes of T' in the i-th pass to construct the m sequences, provided the positions of the leaf nodes of P' in each of the m sequences are known. That, anyway can be calculated in $O(\log n)$ time with $O(n)$ processors once and for all. Now, all these processors simultaneously use binary search to find out the presence (absence) of their corresponding number in the sorted sequence of renumbered nodes of T'. This searching part takes $O(n \log m)$ time. ■

Definition 2.2 *Let $\{n_i\}$ be the set of match nodes in* T *w.r.t.* P *and let $\{l_k\}$ be the set of match nodes in T' w.r.t. P', where $1 \leq i,k \leq m$.*

Lemma 2.12 *The set of match nodes $\{n_i\}$ is a subset of match nodes $\{l_k\}$ and the converse also holds.*

Proof: In general, if $s \in T$ is a match node for P, then let Image(x) = y w.r.t. s, where $x \in P$, $y \in T$ and x is a leaf node. From the definition of TPM, it follows that path(root of P, x) = path(s,y). When P and T are converted into P' and T' respectively, this path information is basically uniquely encoded into a binary number. This in turn implies that path(root of P',renumbered x) is identical to path(renumbered s,renumbered y). This is true for all leaf nodes of P. Hence, by prefix property renumbered node s is still a match node for P'. By an exactly similar argument it can be shown that the converse also holds. ■

Theorem 2.2 *The algorithm TPM_2 finds all match nodes l_k in*

1. $O(\log m)$ *time using* $O(mn)$ *processors.*
2. $O(n \log m)$ *time using* $O(m)$ *processors.*

Proof: As we have already shown, steps 1 through 5 of algorithm TPM_2 can be done either in $O(\log m)$ time with $O(nm)$ processors or in $O(n \log m)$ time with $O(m)$ processors on a CREW PRAM. ■

Chapter 3

Smallest augmentation for biconnecting a graph

3.1 Introduction

In a sense, graph augmentation problem defines the reverse of graph connectivity problem. In the latter case, one would seek to find the minimum number of edges to be deleted from a given graph to render it disconnected. Graph augmentation, on the other hand refers to addition of edges to a graph G in order to make it satisfy some predetermined connectivity property. The connectivity property being sought may refer either to edge connectivity or to vertex connectivity. There will be distinct minimum augmetations to a graph depending upon whether the requirement is for vertex connectivity or edge connectivity. Several polynomial time algorithms are known for the problem of finding a minimum augmentation for a graph to reach some given edge connectivity requirements, on both directed and undirected graphs. It may be noted that edge connectivity refers to the maximum number of edge disjoint paths between every two vertices in the graph. On the other hand, vertex connectivity refers to the maximum number of vertex disjoint paths between every pair of vertices. A polynomial time algorithm has been developed to compute the minimum number of edges to be added to a given graph G so that in the resulting graph the edge connectivity between every pair u, v of its nodes is at least a prescribed value [11]. A polynomial time algorithm to find a set of minimum number of edges the addition of which

makes a graph $2-$ *edgeconnected* is discussed in [9]. Generalizing this for arbitrary $k \geq 2$, a polynomial time algorithm to find a minimum number of new edges to be added to make a graph $k-$ *edgeconnected* is discussed in [37].

The graph augmentation problem for vertex connectivity was solved by Eswaran and Tarjan [9] when the connectivity required is $k = 2$, and the case $k = 3$ by Watanabe and Nakamura [38]. For directed graphs, Eswaran and Tarjan [9] provided a technique to make the graph strongly connected by adding a minimum number of new directed edges. They also proved that the minimum cost version of the problem is *NP-complete*, as the directed Hamiltonian circuit problem can be formulated in this way.

Gusfield [16] proposed a linear time algorithm to augment a mixed graph strongly connected by adding a minimum number of new directed edges. A *mixed graph* consists of both directed and undirected edges. Such a graph is said to be *strongly connected* if, for every pair of nodes u, v there is a path from u to v that consists of directed edges in the right direction and arbitrary number of undirected edges.

There is no known polynomial-time algorithm for finding a smallest augmention to $k-vertex-connect$ a general graph for $k > 3$. There is also no efficient parallel algorithm known to find a smallest augmentation to $k-vertex-connect$ a graph for $k \geq 2$.

Tarjan and Eswaran [9] have provided a characterization of the minimum number of edges required to make an undirected graph biconnected. They have also indicated that a linear time sequential algorithm can be designed for biconnected augmentation problem. However, implementation of that algorithm requires involved list processing. Recently, Hsu and Ramchandran [19] have proposed an $O(\log^2 n)$ time algorithm to biconnect a graph. It requires a linear number of processors on an EREW PRAM. It has also been mentioned that the algorithm can be made to run within the same time bound using a sublinear number of processors. In this paper, we propose a simple $O(\log n)$ time parallel algorithm for the above problem on a CREW PRAM using $O(n)$ processors. This algorithm is based on a direct analysis of block cutpoint tree of the given undirected graph.

3.2 Definitions and notations

Let $G = (V, E)$ be an undirected graph. A vertex $a \in V$ is called an articulation point or a cutpoint if and only if there exist a pair of vertices $u \neq v \in V$ distinct from a such that every path between u and v passes through a. This implies that when a cutpoint $a \in V$ is removed along with all the edges incident upon it, the graph $G' = (V - \{a\}, E')$, where $E' = E - \{(a, x) | x \in V\}$ becomes disconnected. Clearly, the graph G is biconnected if it does not have any cutpoints. A maximal biconnected subgraph of G is called a biconnected component or a block of G. An algorithm for finding biconnected components in a given graph G has been developed by Tarjan and Vishkin [33]. The algorithm achieves a logarithmic running time.

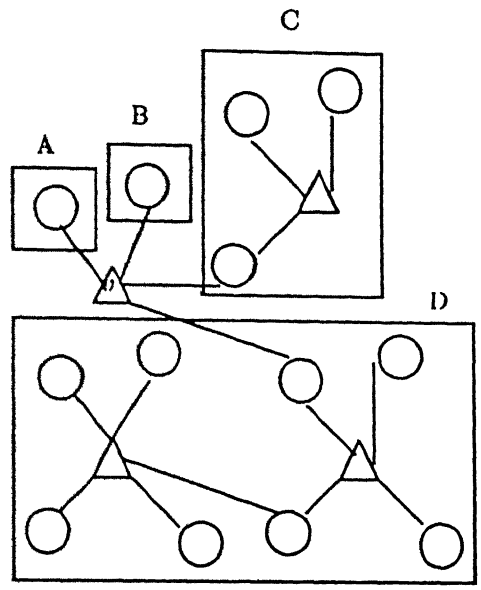
Let G be the graph we wish to make biconnected by adding minimum number of edges. G may have a number of blocks. A block is called *isolated* if it contains no cutpoints, and *pendant* if it contains exactly one cutpoint. Let p and q be the total number of *pendant* and *isolated* blocks of G respectively.

Definition 3.1 *The set of vertices of a block-cutpoint tree* T *of a graph* G *are blocks and cutpoints of* G. *There is an edge between a block-vertex and a cutpoint-vertex in* T *if the corresponding block of* G *contains that cutpoint.*

Definition 3.1 implies that a block-cutpoint tree T is bipartite in which the block vertices and cutpoint vertices occur alternately. The two concepts that will prove to be useful in the analysis of block-cutpoint tree and the intermediate graphs(as more and more edges are added to the original block-cutpoint tree) are defined below.

Definition 3.2 *A group is a maximal collection of cutpoints and blocks with respect to a cutpoint v, such that upon removal of the edge between v and the group in the concerned graph, the group becomes a connected component by itself. The cutpoint v is not an element of any group.*

Illustration of groups is given in Figure 3.1.



Triangles represent cutpoint-vertices
Circles represent block-vertices
A,B,C,D, are the groups with respect to v

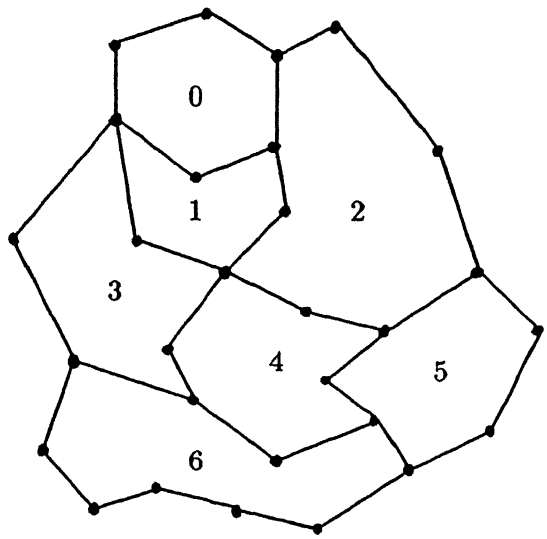
Figure 3.1: Illustration of groups in a block cutpoint tree

Definition 3.3 *A class is a collection of two or more groups such that upon removal of a single cutpoint, there does not exist a path from a member of the class to any vertex outside the class. Classes are defined with respect to cutpoints on the cycle created by addition of first edge to the block cutpoint tree.*

Definition 3.4 *The cardinality of a group is the number of pendant blocks in that group.*

Let there be k_i groups of cardinality i, $1 \leq i \leq l$, in the block cutpoint tree and l is the maximum cardinality with respect to v.
Illustration of classes is given in Figure 3.3.

Let v be any vertex of G. We divide the edges of G into equivalence classes E_i with respect to v such that two edges (x,y) and (w,z) are in the same equivalence class if and only if there exists a path $(v_1,v_2),...,(v_{k-1},v_k)$ containing (x,y) and (w,z) but $v_i \neq v, \forall i, 2 \leq i \leq k-1$. Each equivalence class E_i defines a subgraph $G_i = (V_i, E_i)$ of G, where $V_i = \{x \in V | (x,y) \in E_i\}$. The subgraph G_i is referred to as the v blocks of G. Let $d(v)$ be



Ear decomposition of a biconnected graph
The ears are numbered from 0 through 6

Figure 3.2: Ear Decomposition

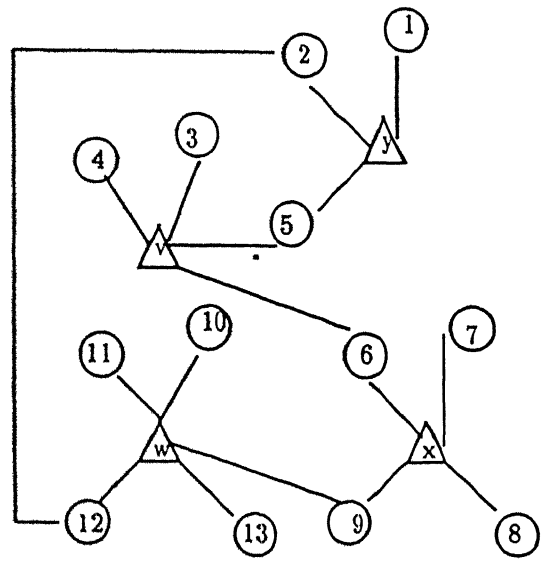
the number of v blocks of G. Then $d(v) > 1$ iff v is a cutpoint. If G contains n connected components and v is contained in b blocks, then $d(v) = b+n-1$. Let $d = max\{d(v)|v \in V\}$.

An ear decomposition, denoted by $D = \{P_0, P_1, ...P_{r-1}\}$, of a biconnected undirected graph is a partition of the set of edges E into an ordered collection of edge disjoint simple paths $P_0, P_1, ...P_{r-1}$ called ears, such that P_0 is a simple cycle and each P_i for $i(= 1, 2, ..., r-1)$ is a simple path (possibly a simple cycle) with its two end vertices belonging to a lower numbered ear, but no internal vertices of the path belong to any lower numbered ear. An ear decomposition of a biconnected is shown in Figure 3.2. An open ear decomposition is an ear decomposition in which none of P_i is a cycle. Note that a graph G has an open ear decomposition iff it is biconnected [25]. This observation is crucial to our algorithm.

3.3 Biconnected Augmentation

Theorem 3.1 *$max(d-1, \lceil p/2 \rceil + q)$ edges are necessary and sufficient to make* G *biconnected if $p+q > 1$.*

Proof: Refer [9]. ■



x is root of block cutpoint tree
the arc between blocks 2 & 12
is the first edge added

blocks 3 & 4 form a class w.r.t. v
blocks 10, 11 & 13 form a class w.r.t. w
blocks 7 & 8 form a class w.r.t. x

Figure 3.3: Illustration of classes in a block cutpoint tree of Figure 3.1

To start with, construct a block-cutpoint tree for G. This can be done in $O(\log n)$ time using $O(n)$ processors [33].

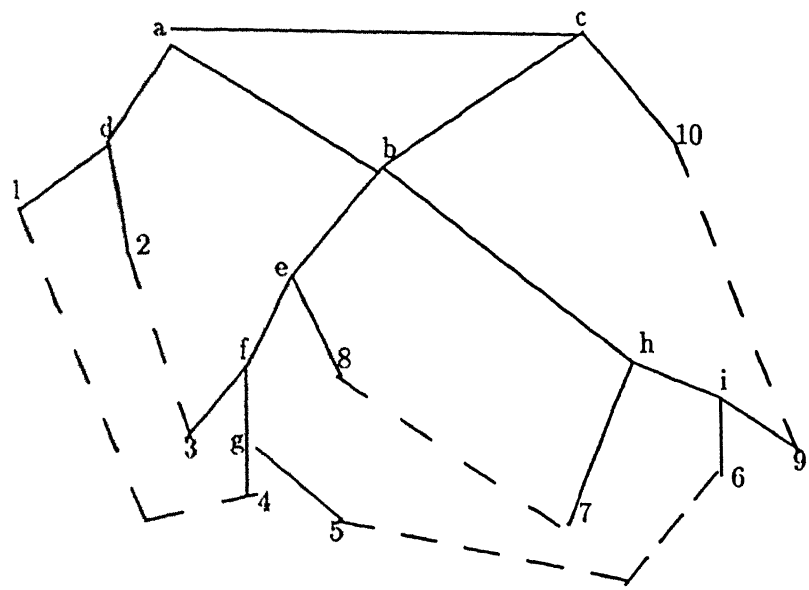
For sake of convenience, assume G is connected (i.e. $q = 0$). If this is not the case, G can be made connected by adding $q - 1$ edges.

The quantities d and p can be calculated from the block-cutpoint tree by using any of the tree traversal techniques. The next step involves rooting the block-cutpoint tree at a suitable node, so that the size of none of the groups formed upon addition of the first edge exceeds $\lceil p/2 \rceil$. To do this, the optimal parallel algorithm for expression tree evaluation [24] may be used. Using that, any arithmetic expression of size n can be evaluated in $O(\log n)$ time with $O(n/\log n)$ processors on an EREW PRAM. Besides, the same algorithm when modified slightly can evaluate every subexpression of the original arithmetic expression. Before we can apply this algorithm, the block-cutpoint tree is to be converted to a strict binary tree. A strict binary tree is one in which every internal node has exactly two children. Any tree can be converted to a strict binary tree by the addition of $O(n)$ dummy nodes. In expression trees, every internal node stores an operator (+ in our case) and each of the leaf nodes store a constant. The constants asociated with the leaf nodes (pendant blocks) are a 1, and the dummy nodes a 0. Evaluating all subexpressions will yield the number of descendant pendant blocks for each of the internal nodes. If a block other than the root has more than $\lceil p/2 \rceil$ pendant blocks, then shift the root to this particular block. Three different cases arise when we compare $d - 1$ with $\lceil p/2 \rceil$. These cases are analysed and handled appropriately.

3.3.1 Case 1: $d - 1 > \lceil p/2 \rceil$.

It is clear that there can be exactly one cutpoint v with $d(v) = d$.

The basic strategy now is to repeatedly connect two pendant blocks via a new edge so that the resulting augmented graph is biconnected. For this construct a binary tree of the groups of cardinality greater than 2 by adding new edges as shown in Figure 3.5. Every new edge will connect two pendant blocks belonging to different groups. Note that the first edge introduced will result in a closed ear(a simple cycle) and every subsequently edge added results in an open ear(when two pendant blocks belonging to different classes are

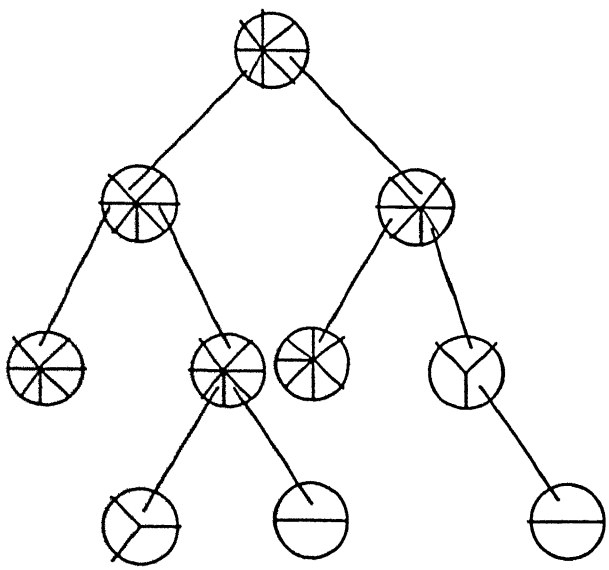


An example for biconnecting a graph
The numbered vertices represent the pendant blocks
Dashed lines indicate the edges added to the graph
The vertices named a through i are the cutpoints

Figure 3.4: Reorganization of ears

connected via a new edge) or in a closed ear(when two pendant blocks belonging to the same class are short-circuited). However, in the latter case, as shown in figure 3.4, there is way to reorganize the closed ears so formed to get rid of the closed ears. As shown in Figure 3.4, the closed ear formed by b, e, f, g, 5, 6, i, h, b is taken as the open ear b, e, f, g, 5, 6, i, 9, 10, c and another open ear b, h, i. The cutpoints a, b, c form the first cycle formed and also the only closed ear. Since one pendant block is always left in a group to be matched to another pendant block outside that class, this reorganization process is always possible. This ensures that the final augmented graph is biconnected. The edges added in this manner eventually decide how to augment G to make it biconnected.

Clearly, $x = \sum_{i=3}^{i=l} k_i - 1$ edges are needed for construction of this binary tree, using up twice this number of pendant blocks. The height of this tree is $\Omega(n)$ and can be constructed in $O(\log n)$ time using $O(n)$ processors. Groups of cardinality 2 are incorporated in this binary tree using an additional k_2 edges, thus leaving the number of pendant blocks in the augmented graph unchanged.



Groups are represented by circles and pendant blocks by partitions within the circles

Figure 3.5: Binary tree construction

Lemma 3.1 *On addition of groups of cardinality 2 into the augmented graph, it contains* $y = p - 2(x + k_2) - k_1$ *pendant blocks. Moreover,* $k_1 > y$.

Proof: The first assertion is quite straightforward. It follows from the fact that every edge of the binary tree built out of the groups of the block cutpoint tree reduces the number of pendant blocks by 2 as shown in Figure 3.5. If the second condition is not true, it leads to a contradiction of theorem 3.1, as it implies that G can be made biconnected using less than $d - 1$ edges. ■

Lemma 3.2 *The total number of edges required to make G biconnected by using the above mentioned method is* $d - 1$.

Proof: The number of edges used so far is $x + k_2$. An additional k_1 edges are required as $k_1 > y$ from Lemma 3.1. Thus, in order to make G biconnected a total of $x + k_2 + k_1$ edges are required. Note that $(\sum_{i=1}^{i=1} ik_i) = p$ and $x + 1 + k_1 + k_2 = d$. The result follows from these observations. ■

3.3.2 Case 2: $d-1 = \lceil p/2 \rceil$.

There can be atmost two vertices which satisfy the equation $d-1 = \lceil p/2 \rceil$. This case is handled as in case 1. The number of edges required is $d-1$.

3.3.3 Case 3: $d-1 < \lceil p/2 \rceil$.

Two cases arise depending upon the sizes of the classes (i.e. number of pendant blocks within a class).

Case(a). All classes are of sizes less than $\lceil p/2-1 \rceil$. Let the sizes of the classes be $k_1, k_2, ..., k_x$.

1. Find prefix sums $S_i = \sum_{m=1}^{m=i} k_m, 1 \leq i \leq x$ [7].
2. Use binary search to find the smallest i, such that $S_i \geq \lceil p/2 \rceil$.
3. If $S_i > \lceil p/2 \rceil$, split the members of the ith class into two parts k_i^a and k_i^b, so that $\sum_{j=1}^{j=i-1} k_j + k_i^a = \lceil p/2 \rceil$. Let A^* represent the members of $\sum_{j=1}^{j=i-1} k_j + k_i^a$ and B^* represent the rest of the pendant blocks.
4. Match members of A^* with those of B^*, with the restriction that k_i^a do not get matched with k_i^b.

It is clear that G becomes biconnected through addition of $\lceil p/2 \rceil$ edges.

Case(b). There exists a class of size greater than $\lceil p/2 - 1 \rceil$.

1. Build a binary tree of that many groups (which are members of this class) so that the class size reduces to less than half of the remaining pendant blocks in the augmented graph resulting from block cutpoint tree.
2. Match as in step 4 of Case(A)

Theorem 3.2 *The graph* G *can be augmented by adding minimum number of edges in* $O(\log n)$ *time using* $O(n)$ *processors.*

■

Chapter 4

Triconnectivity Augmentation

4.1 Introduction

The generalization of the concepts involved in biconnectivity and biconnected augmentation i.e., $k - vertex$ connectivity ($k > 2$) for graphs is studied in this chapter. A graph is said to be $k - vertex$ connected if the minimum number of vertices to be deleted from G so that the resulting graph is disconnected, is k. In particular, we focussed our attention to the problem of augmenting a given graph by addition of a minimum number of edges to make it triconnected. In a triconnected graph G, at least three vertex disjoint paths exist between every pair u, v of vertices of G.

An algorithm for finding triconnected components of a graph is useful in many areas such as analyzing electrical networks, planarity testing of a graph and for determining whether two planar graphs are isomorphic. Hopcroft and Tarjan [18] proposed an $O(|V| + |E|)$ time sequential algorithm for this purpose. Recently Fussel and Ramchandran [12] have given an $O(\log n)$ time parallel algorithm for finding triconnected components, where m and n are number of edges and vertices in the graph. This algorithm is based upon the idea of open ear decomposition. The model of computation used is CRCW PRAM and the work done by the algorithm is $O((m+n)\log\log n)$. Miller and Reif [28] have proposed another $O(\log n)$ time parallel algorithm for computation of the tree of $3 - connected$ components of G using $n^{O(1)}$ processors.

The tree of $3-$ *connected* components consists of a tree of graphs called components. Two components are adjacent if they share an edge. One of their assumptions is that G is biconnected. Their approach uses the idea of parallel tree contraction [6, 24, 27]. Miller and Reif have also tackled the problem the embedding of a planar graph, given its tree of $3-$ *connected* components.

The problem of finding a smallest augmentation to triconnect a graph G, was solved separately by Watanabe [36] and Hsu and Ramchandran [20]. The sequential algorithm presented in [36] has a time complexity of $O(n(n+m)^2)$. Hsu and Ramchandran improved the bound to linear time. In the approach used by Hsu and Ramchandran [20], there are two distinct phases. During the first phase, the given graph is biconnected using a smallest number of edges. In the second stage, the resulting biconnected graph is triconnected by inserting the minimum number of edges. Since there exists a graph G such that any smallest augmentation for biconnecting G does not lead to a smallest augmentation for triconnecting G, some care has to be taken in stage one. This peculiar problem does not arise for edge-connectivity because of a result by Naor, Gusfield and Martel [30]. They have shown that there exists a smallest augmentation to $k-edge-connect$ a graph G such that it is included in a smallest augmentation to $(k+1)-edge-connect$ G, for an arbitrary k.

This chapter is an investigation into the specific problems and difficulties enocuntered in our effort to design a parallel algorithm to the problem of finding the smallest augmentation to make a given graph G triconnected. The $3-blk(G)$ see section 4.2, at a cursory look, is quite similar to the block-cutpoint tree for biconnected graphs. However, there are many intricate details and difficulties involved which call for some more insight into the structure and properties of $3-blk(G)$. Some of these properties are mentioned in [20]. We tried the divide and conquer technique in which a few of the required set of edges are found in every stage. The graph G and $3-blk(G)$ have to be modified appropriately after each stage. However, a satisfactory way to arrive at intermediate $3-blk(G)$ trees could not be found out, though we still feel that under the above framework a procedure can be worked out. This problem of triconnectivity augmentation of any graph in parallel is open to research. We, therefore looked at the problem for a restricted class of graphs called outer planar graphs. Outer planar graphs, as the name suggests, are planar graphs with the property

that all the vertices lie on the outer face. It was found that this problem renders itself to an easy parallel solution.

4.2 Definitions and Notations

Let $G = (V, E)$ be an undirected graph. Corresponding to the concept of articulation points in case of biconnected graphs, separation pairs are defined here. Let $\{a, b\}$ be a pair of vertices of G.

Definition 4.1 *A separation class $E_i, 1 \leq i \leq n$, consists of edges which lie on a common path not containing any vertex of $\{a, b\}$ except as an endpoint.*

Clearly, separation classes define equivalence classes on the edges of G. If there are atleast two separation classes with respect to $\{a, b\}$, then $\{a, b\}$ is called a separation pair of G unless:

1. There are exactly two separation classes, and one class consists of a single edge.

2. There are exactly three classes, each consisting of a single edge.

If G is a biconnected graph with no separation pairs, then G is triconnected.

Let the separation classes of G with respect to $\{a, b\}$ be $E_i, 1 \leq i \leq n$. Let $E' = \bigcup_{i=1}^{k} E_i$ and $E'' = \bigcup_{i=k+1}^{n} E_i$ be such that $\mid E' \mid \geq 2, \mid E'' \mid \geq 2$. Let $G_1 = (V(E'), E' \bigcup \{(a, b)\})$ and $G_2 = (V(E''), E'' \bigcup \{(a, b)\})$. The graphs G_1 and G_2 are called the split graphs of G with respect to $\{a, b\}$. The operation of replacing a graph G by two split graphs is called splitting G. There may be many different ways to perform this splitting operation, even with respect to a fixed separation pair. If G is biconnected, then any split graph of G is also biconnected. When a graph G is split, the split graphs are split, and so on, until no more splits are possible, each of the remaining graphs is trconnected. The graphs constructed in this way are called the split components and the split components of a graph are not necessarily unique. In order to get unique triconnected components, the split components must be partially reassembled.

Analogous to block-cutpoint tree for biconnected graphs, a $3 - blockgraph$, denoted by $3 - blk(G)$ is defined. The $3 - blockgraph$ contains three sets of vertices:

- For every triconnected component of G (including trivial ones), a β -vertex is present in the $3 - blk(G)$.
- A π -vertex in $3 - blk(G)$ represents a simple cycle of G.
- If v is a vertex in a simple cycle A with degree 2 and if x and y are the two vertices in A that are adjacent to v, then $\{x, y\}$ form a separation pair and have a corresponding σ-vertex.

Vertices u and v in $3 - blk(G)$ are adjacent if any of the following conditions is true:

1. u is a β-vertex, v is a σ-vertex and the triconnected component corresponding to u contains the pair of vertices corresponding to v.
2. u is a β-vertex corresponding to a degree-2 vertex w in G and v is the σ-vertex corresponding to the pair of vertices in G that are adjacent to w.
3. u is a π-vertex corresponding to a simple cycle A in G, v is a σ-vertex corresponding to a pair of vertices b_1 and b_2 in A.

From [18, 32], we know that $3 - blk(G)$ is a tree if G is biconnected. This tree is called a $3 - block - tree$, and the set of trees corresponding to biconnected components in G the $3 - blockgraph$ of G or $3 - blk(G)$. Given a graph G with n vertices and m edges, the $3 - blk(G)$ can be computed in $O(n + m)$ time using procedures in [18]. An example of a graph and its corresponding $3 - blk(G)$ tree is given in figure 4.1.

Given a σ-vertex s, let (a_1, a_2) be the corresponding pair of vertices, define $d_3((a_1, a_2))$ or $d_3(s)$ to be the degree of s in the $3 - blockgraph$. Note that $d_3((a_1, a_2)) \geq 2$ for any separation pair. Taking $d_2(v)$ to be the degree of v in block-cutpoint tree of G, d_2 is 1 if v is not a cutpoint. Given a graph with k connected components, the *separation degree*

$$sd((a_1, a_2)) = \sum_{i=1}^{2} d_2(a_i) + d_3((a_1, a_2)) + h - 4$$

Here, h is the number of connected components of G. It is shown in [20] that *separation degree* equals the smallest number of edges needed to connect the graph obtained from G by removing a separation pair.

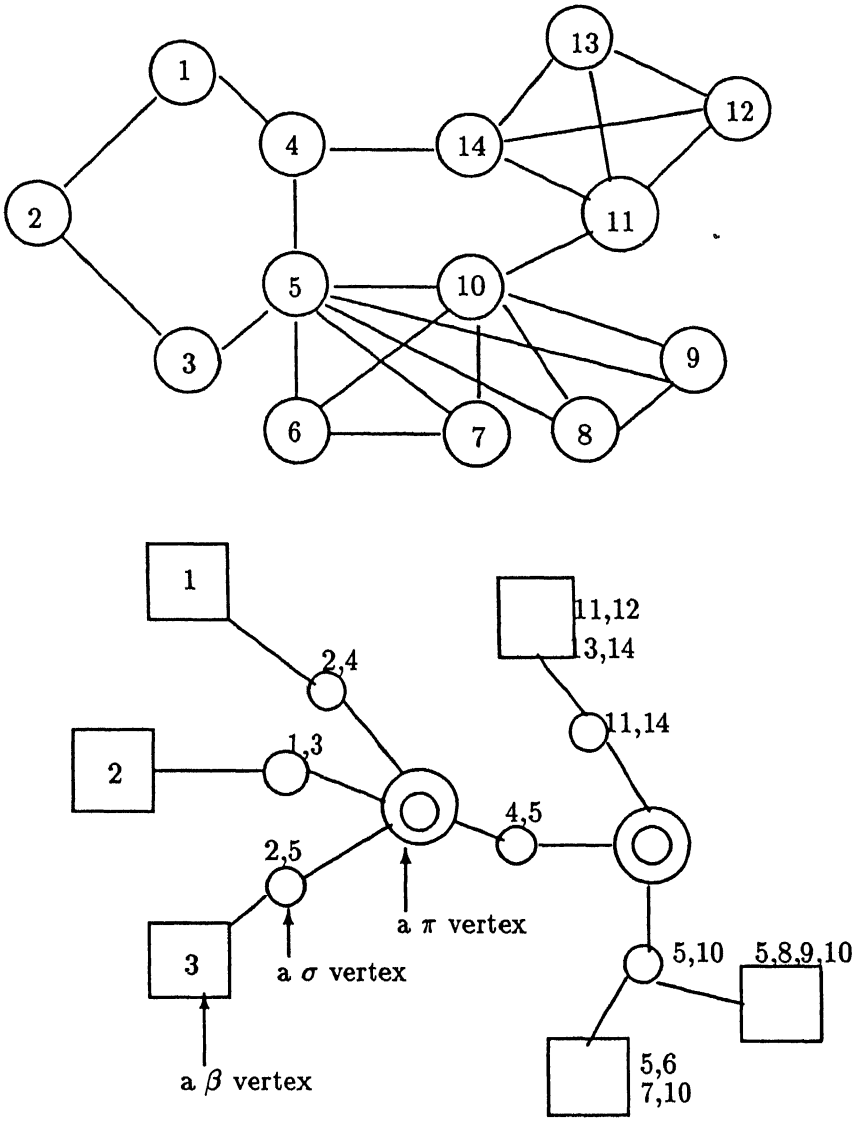


Figure 4.1: A graph G and its $3 - blk(G)$

Given a leaf node or an isolated $3-block$ vertex b in $3-blk(G)$, let the corresponding triconnected component be G_b. A vertex u in G_b is a demanding vertex of b if any of the following conditions is true:

1. u is not a cutpoint or in any of separation of G contained in G_b.
2. b is a isolated $3-block$ vertex and G_b consists of only the vertex u. The vertex u is also called a demanding vertex of G.

Let $l_3(G)$ be the number of $3-block$ leaves in G.

Theorem 4.1 *Given an undirected graph* G, *at least* $max(d, \lceil l_3(G)/2 \rceil)$ *edges are required to triconnect* G, *where* d *is the largest separation degree among all* σ*-vertices in* $3-blk(G)$.

Proof: Refer [20].

4.3 Triconnected Augmentation of an Outer Planar Graph

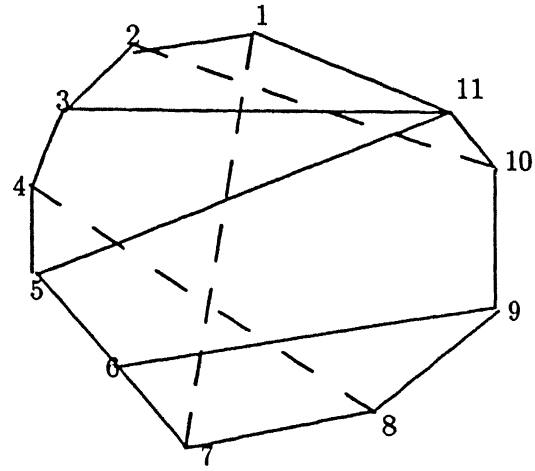
Finding a smallest set of edges to triconnect an outer planar graph is quite straightforward. Let the vertices be numbered $1, 2, \ldots, n$ in the order that they appear on the outer face. Let the vertices of degree 2 be $k_1, k_2, \ldots, k_l$, where $1 \leq k_i \leq n$, for $1 \leq i \leq l$.

- Sort the numbers $k_i, 1 \leq i \leq l$.
- If l is even, the new edges to be inserted are (k_2, k_l), $(k_3, k_{l-1}), \ldots, (k_{l/2-1}, k_{l/2+1})$, $(k_1, k_{l/2})$.
- If l is odd, the new edges to be inserted are (k_2, k_l), $(k_3, k_{l-1}), \ldots, (k_{(l+1)/2}, k_1)$, $(k_{(l+3)/2}, k_1)$.

An example of triconnecting an outer planar graph is shown in Figure 4.2.

Let us call a newly inserted edge (x, y) as a *latitude* when x and y are distinct from k_1, otherwise such an edge is known as a *longitude*.

Theorem 4.2 *All separation pairs which are also the end-points of singular bridges will no longer be speration pairs upon addition of the latitudes and longitudes to the original graph.*



Solid lines show the edges of the outer planar graph
Dashed lines show the edges inserted

Figure 4.2: Triconnecting an outer planar graph

Proof: A singular bridge (a, b) in G divides the vertices of G into two sets:

- Vertices on the path from a to b on the outer face, when the outer face is traversed in clockwise direction. Let X denote this set.
- Vertices on the path from a to b on the outer face, when the outer face is traversed in anti-clockwise direction. Let Y denote this set.

It is clear that both X and Y must contain vertices of degree 2. We know that the minimum number of edges required to triconnect G is half the number of degree 2 vertices in G. When an edge(latitude or longitude) joins a vertex from X to a vertex belonging to Y, the pair (a, b) is no longer a separation pair. Following our scheme for addition of edges, this is bound to happen.

It may so happen that degree 2 vertices of X and Y are matched together, but at least one such degree 2 vertex will remain in both X and Y. These two vertices will be connected ultimately via a latitude or a longitude. This completes the proof. ■

It may be noticed that time required for inserting new edges as stated above is dominated by sorting of l numbers. The other step can be done in $O(1)$ time in parallel when $O(n)$ processors are available. Since sorting of n elements can be done in $O(\log n)$ time with

$O(n/\log n)$ processors [4], the whole algorithm to find a smallest augmentation to 3–*connect* an outer planar graph requires $O(\log n)$ time with $O(n/\log n)$ processors on PRAM.

Chapter 5

Summary and conclusions

In this thesis three different problems have been discussed. The first parallel algorithm for Restricted Tree Pattern Matching based on tree merging as proposed in chapter 2 is not cost optimal. Another closely related and simple approach to RTPM could be through splitting of target tree T into a forest. It can be viewed as a single step inverse of the process of tree merging. Let the height of a node v in T be defined as equal to the maximum depth of all descendants of v in T. In tree splitting approach, first height of v, for all v in T, is computed. Then T is split into a forest of trees by declaring the parents of all the nodes, whose height equals the height of P, as null. Note that in this forest, each node v in T is present in at most one of the trees. The roots of the trees in the forest are the only potential match-candidates. Actual matching is determined by executing steps 5 and 6 as done in the tree merging approach.

Height of every node in the target tree T can be computed in $O(\log m)$ time, using $O(m/\log m)$ processors deterministically on an EREW PRAM. This is done by reducing this problem to the standard problem of dynamic parallel evaluation of computation trees, which in turn can be solved in the stated bounds [29]. The given arbitrary target tree T is converted to an equivalent regular binary tree [22] by adding $\Omega(m)$ dummy nodes. The operators associated with the internal nodes are max, with the exception that operators at the internal nodes with a superscript 1 are $max(x, y) + 1$. The leaf nodes are labelled with zeros. The values subsequently calculated at the internal nodes are the heights of these nodes.

The second parallel algorithm for RTPM proposed in chapter 2 is based upon Vishkin's pattern matching algorithm. This is a cost optimal algorithm. The advantage of the coding (for the tree) suggested here, over the work of Grossi [15] is that it could ensure isomorphism checks along with data associated with each node. This could be achieved by associating a data field with every left bracket corresponding to a node. The string matching could be performed by first matching the parenthesis field and then (if successful) the data string field corresponding to the node.

Algorithm TPM_2 is the simplest yet the best among all three solutions outlined in chapter 2. The tree pattern matching problem discussed in chapter 2, as yet does not have a cost optimal parallel algorithm . The best known sequential algorithm has a time complexity of $O(mn^{0.5}polylog(n))$. The parallel algorithm we have proposed has a time complexity of $O(\log m)$ using $O(mn)$ processors. The lower bounds on the sequential complexity have improved over the years. We believe that a better parallel approach for locating the match-nodes in the target tree is possible. As for the restricted tree pattern matching, there is scope for further research. The cost-optimal algorithm we have given is based upon Vishkin's [35] pattern matching algorithm, which in turn reqiures the use of a CRCW PRAM. Whether the same cost bounds are achievable on weaker CREW or EREW PRAMs remains an open problem.

In case of biconnected augmentation, though we have given an algorithm which matches the performance of another algorithm by Hsu and Ramchandran [20], it is not cost-optimal. Note that all CREW PRAM algorithms can be simulated on an EREW model with an additional time factor of $O(\log n)$. The overall pattern of our solution looks similar to the algorithm proposed in [20], but there are some significant differences. We have relied heavily on the connection between open ears and biconnectivity. Designing a cost-optimal parallel algorithm for the task remains an open problem.

For the triconnectivity augmentation problem for graphs, the existing sequential algorithms present a conceptual framework of the issues involved. To arrive at a parallel algorithm, the structure of $3 - blk(G)$ has to be thoroughly explored. It is worth mentioning here that there is no polynomial time algorithm known for finding a smallest augmentation to $k - vertex - connect$ a graph for $k > 3$.

Bibliography

[1] Anderson R. and Miller G., Deterministic parallel list ranking, *Lecture notes in Computer Science*, Vol. 319, (Springer, Berlin, 1988), pp. 81-90.

[2] Carter J.L. and Wegman M.N., Universal class of hash functions, *Proc. 9th Ann. ACM Symp. on Theory of Computing*, 1977, pp. 106-112.

[3] Chen C.Y., Das S. and Akl S., A Unified Approach to Parallel Depth-First Traversals of General Trees, *Infor. Proc. Lett.*, 38(1991), pp. 49-55.

[4] Cole R., Parallel Merge Sort, *SIAM J. Comput.*, 2(1988), pp. 770-785.

[5] Cole R. and Vishkin U., Approximate and exact parallel scheduling with application to list, tree, and graph problems, *Proc. 27th Ann. Symp. on FOCS*, 1986, pp. 478- 491.

[6] Cole R. and Vishkin U., Optimal parallel algorithms for expression tree evaluation and list ranking, *Lecture Notes in Computer Science*, Vol. 319, Springer Verlag, New York, 1988, pp. 91-100.

[7] Cole R. and Vishkin U., Deterministic coin tossing with applications to optimal parallel list ranking, *Inform. and Control*, 76(1986), pp. 32-53.

[8] Dubiner M., Galil Z. and Magen E., Faster Tree Pattern Matching, *Proc. 31st Ann. Symp. on FOCS*, 1990, pp. 145-150.

[9] Eswaran K.P. and Tarjan R.E., Augmentation problems, *SIAM J. Comput.*, 5(1976), pp. 653-665.

[10] Evens S., *Graph Algorithms*, Computer Science Press, 1979.

[11] Frank A., Augmenting graphs to meet edge-connectivity requirements, *SIAM J. Comput.*, 5(1992), pp. 25-53.

[12] Fussell D., Ramchandran V. and Thurimella R., Finding triconnected components by local replacement, *SIAM J. Comput.*, 22(1993), pp.587-616.

[13] Gibbons A. and Rytter W., *Efficient parallel Algorithms*, Cambridge University Press, 1988.

[14] Gibbons P.B., Soroker D., Karp R.M. and Miller G.L., Subtree isomorphism is in random *NC*, *Lecture Notes in Computer Science*, Vol. 319, pp. 43-52.

[15] Grossi R., A Note on the Isomorphism for Ordered Trees and Related Problems, *Infor. Proc. Lett.*, 39(1991), pp. 81-84.

[16] Gusfield D., Optimal mixed graph augmentation, *SIAM J. Comput.*, 16(1987), pp.599-612.

[17] Hoffman C.M. and O'Donnell, Pattern Matching in Trees, *J. of ACM*, 1982, pp. 68-95.

[18] Hopcroft J. E. and Tarjan R. E., Dividing a graph into triconnected components, *SIAM J. Comput.*,, 2(1976), pp. 135-158.

[19] Hsu T. and Ramchandran V., Finding a smallest augmentation to biconnect a graph, *SIAM J. Comput.*, 22(1993), pp. 889-912.

[20] Hsu T. and Ramchandran V., A linear time algorithm for triconnectivity augmentation (extended abstract), *manuscript*, 1991.

[21] Kedem Z.M. and Palem K.V., Optimal parallel algorithms for forest and term matching, *Theo. Computer Sc.*, February 1992, pp. 245-264.

[22] Knuth D., *The art of computer programming, Vol.1, Fundamental Algorithms*, Addison-Wesley, 1968.

[23] Kosaraju S.R., Efficient Tree Pattern Matching, *Proc. 30th Ann. Symp. on FOCS*, 1989, pp. 178-183.

[24] Kosaraju S.R. and Delcher A.L., Optimal parallel evaluation of tree-structured computations by raking (extended abstract), *Lecture Notes in Computer Science*, Vol.319, pp. 101-110.

[25] Maon Y., Scheiber B. and Vishkin U., Parallel ear decomposition search (EDS) and st-numbering in graphs, *Theoretical Computer Science*, 47(1986), pp. 277-296.

[26] Miller G.L. and Reif J.H., Parallel tree contraction and its applications, *Proc.26th Ann. Symp. on FOCS*, 1985, pp.478-489.

[27] Miller G.L. and Reif J.H., Parallel tree contraction Part 1: Fundamentals, in *Randomness and Computation*, Vol. 5, JAI press, Greenwich, CT, 1989, pp. 47-72.

[28] Miller G.L. and Reif J.H., Parallel tree contraction Part 2: Further applications, *SIAM J. Comput.*, 20(1991), pp. 1128-1147.

[29] Miller G.L. and Teng S., Systematic methods for tree based parallel algorithm development, *Second International Conference on Supercomputing*, 1987.

[30] Naor D., Gusfield D. and Martel C., A fast algorithm for optimally increasing the edge-connectivity, *Proc. 31st Ann. Symp. on FOCS*, 1990, pp. 698-707.

[31] Rajasekaran S. and Reif J.H., Optimal and sublogarithmic time randomized parallel sorting algorithms, *SIAM J. Comput.*, 18 (1989), pp. 594-607.

[32] Ramchandran V., Parallel open ear decomposition with applications to graph biconnectivity and triconnectivity, invited chapter for *Synthesis of Parallel Algorithms*, J.H. Reif, editor, Morgan-Kaufmann.

[33] Ramesh R., Verma R.M., Krishnaprasad T., Ramkrishnan I.V., Term matching on parallel computers, *Proc. 14th ICALP*, 1987.

[34] Tarjan R.E. and Vishkin U., Finding biconnected components and computing tree functions in logarithmic parallel time, *Proc. 25th Ann. Symp. on FOCS*, 1984, pp. 12-22.

[35] Tsin Y.H. and Chin F.Y., Efficient parallel algorithms for a class of graph theoretic problems, *SIAM J. Comput.*, 13 (1984), pp. 580-598.

[36] Vishkin U., Deterministic Sampling - A new technique for fast pattern matching, *SIAM J. Comput.*, 20 (1991), pp. 20-40.

[37] Watanabe T. and Nakamura A., A smallest augmentation to 3 – *connect* a graph, *Discrete Appl. Math.*, 28(1990), pp. 183-186.

[38] Watanabe T. and Nakamura A., Edge-connectivity augmentation problems, *J. Comp. System Sci.*, 35(1987), pp. 96-144.